Assisting Teaching Assistants with Automatic Code Corrections

Yana Malysheva
Caitlin Kelleher
Washington University in St. Louis
St. Louis, Missouri, USA

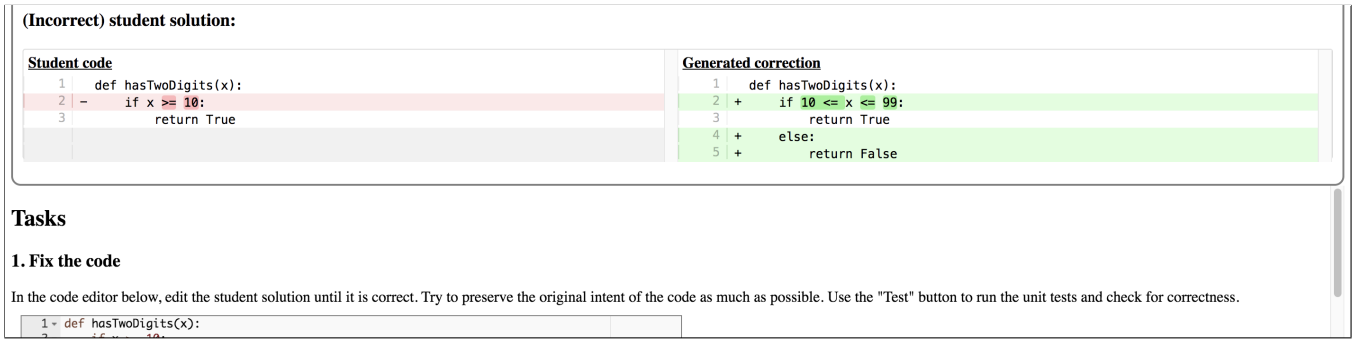

Figure 1: A fragment of the debugging interface used in the study

ABSTRACT

Undergraduate Teaching Assistants(TAs) in Computer Science courses are often the first and only point of contact when a student gets stuck on a programming problem. But these TAs are often relative beginners themselves, both in programming and in teaching. In this paper, we examine the impact of availability of corrected code on TAs' ability to find, fix, and address bugs in student code. We found that seeing a corrected version of the student code helps TAs debug code 29% faster, and write more accurate and complete student-facing explanations of the bugs (30% more likely to correctly address a given bug). We also observed that TAs do not generally struggle with the conceptual understanding of the underlying material. Rather, their difficulties seem more related to issues with working memory, attention, and overall high cognitive load.

ACM Reference Format:
Yana Malysheva and Caitlin Kelleher. 2022. Assisting Teaching Assistants with Automatic Code Corrections. In *ACM CHI Conference on Human Factors in Computing Systems, April 30 - May 6 2022, New Orleans, LA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3491102.3501820

1 INTRODUCTION

Undergraduate teaching assistants (TAs) are often the first and only point of contact when a student gets stuck on a programming problem as a part of their class assignment. In order to effectively help a student overcome their difficulties and advance their understanding of the subject matter, the TA needs to perform several tasks simultaneously, within a limited amount of time: reason about the student's intentions in writing their code; find the mistakes that prevent the student's current code from fulfilling those intentions; and explain the problem to the student, striving to reinforce the concepts that the original problem was intended to address. Handling all of these disparate tasks at the same time can be challenging even for an experienced educator, but TAs are often new to teaching, and relatively new to programming. Therefore, it is not surprising that they may have difficulties managing all of these responsibilities at once.

To better assist TAs, it is important to understand which aspects of these tasks serve as barriers. Yet, little is known about how TAs approach helping students, where they struggle, and how those struggles impact their interaction with learners.

In this paper, we examine the impact of the availability of auto-generated, corrected code on TA behavior when the TA is engaged in finding and explaining bugs in student code. Seeing a comparison between the buggy student code and a corrected version of the same code (as illustrated in Figure 1) can quickly provide TAs with a sense of the student's intention and an overview of their mistakes. This may enable TAs to invest more time and mental energy in providing a good explanation to the student.

In order to evaluate the effect of seeing the corrected code, we ran a series of studies in which TAs first debugged code written by introductory computer science students, and then wrote a helpful explanation to the student who wrote the code. Results show that TAs who saw a corrected version of the student code completed the tasks faster and more accurately than those who did not.

We also discuss the patterns we observed in TA behavior when they debug student code, the difficulties they tend to have, and what kinds of tools may be able to address these difficulties. We believe that the major source of difficulty in this task comes from high cognitive effort, and associated slips and lapses of attention, rather than a lack of conceptual understanding of the underlying

CHI '22, April 30 - May 6 2022, New Orleans, LA

ACM ISBN 978-1-4503-9157-3/22/04.
https://doi.org/10.1145/3491102.3501820

material. For example, we found that TAs actually have more difficulty finding the second and third instance of the same bug in a short snippet of code than the first instance. If this is true, then tools that help reduce cognitive demands on TAs may be able to improve TAs' impact on student learning.

2 RELATED WORK

This work builds on and contributes to research in four areas: Undergraduate Teaching Assistant (UTA) programs, peer tutoring, debugging, and automatic code correction.

2.1 Undergraduate CS TAs

The existing literature on TAs and learning is small and focused on the evaluation of programs rather than in-depth analyses of TA-student interactions. A recent review of the literature [32] in this area finds that undergraduate TA programs claim benefits that fall into three broad categories: 1) benefits to student learners, 2) benefits to teaching assistants, and 3) benefits to the instructor and organization.

2.1.1 Benefits to student learners. Researchers evaluating the impact of different TA programs have found improvements in student performance [5, 13, 35] and student satisfaction [11, 31].

Biggers et al found that changing recitation sections to be problem solving driven rather than a re-lecture improved student's grade performance [5]. Erdei found that an additional undergraduate TA to support labs was comparable to a peer programming approach in supporting student understanding on a programming assessment [13]. Pivkina found that the availability of peer learning assistants (in addition to the instructor and TAs) improved students' average course score [35].

Both Decker et. al and Minnes et al. found that additional access to undergraduate TAs (UTAs) can improve student experience. Decker et al found that the use of UTAs in CS1 was associated with higher subjective student satisfaction [11]. Minnes found that the availability of individually assigned near-peer tutors improved the sense of community among learners [31].

2.1.2 Benefits to TAs. Several studies have also documented benefits for the students who serve as TAs. Fryling et al. found in focus groups that the experience of serving as a mentor during lecture and lab improved TAs skills and confidence [15] Brent et al. found a a similar boost in teaching confidence for TAs who attended an eight hour training before starting their TA assignment [6]. Finally, [45] found that after serving as a TA, students rated their own interpersonal skills more highly.

2.1.3 Benefits to the instructor/organization. Finally, some research points to benefits for the instructors and organizations offering courses. UTAs can make the operation of large classes with limited staff more feasible [7]. A number of these papers discuss departmental practices in hiring, training, and evaluating TAs [6, 11, 15, 35]. However, we note that the 2019 literature review notes that "we found limited research that links specific practices to benefits and program goals" [32].

Overall, our understanding of TAs is sparse and largely informed by studies of programs. Studies of the interactions between students and TAs could potentially identify opportunities to improve TA efficiency, student learning, and student experience.

2.2 Scaffolding peer tutoring

Several studies explore how best to scaffold interactions between students in a programming or CS course and peer or near-peer assistants who can help students make progress on their tasks. The interventions proposed in these studies target a variety of potential roadblocks to effective peer help. Kos [25] seeks to explicitly scaffold good peer tutor practices through a "Collaborative Learning Framework" poster. Liu et al.[28] and Fong et al.[14] organize and motivate potential peer tutors to provide assistance to students. Glassman et al.[16] describe several ways of organizing peer-created information to be more discoverable and useful to the students who need it. However, while these interventions support TAs in remembering best practices, they do not seek to understand or address barriers that TAs experience.

Outside of the field of computer science, one study[46] does examine the effects of providing real-time problem-specific information to the peer assistant. In this study, students worked through algebra problems using an Intelligent Tutoring System (ITS), either with or without peer help. When peer help was present, the ITS presented student feedback to the peer tutor rather than the student directly, and the peer tutor chose what information to pass on to the student. They found no difference in outcomes, though there were some qualitative differences in how the different groups used the system.

In this work, we explore the effects of providing problem-specific information that is tailored to the target audience of a peer assistant.

2.3 Debugging

There is a large body of research on how programmers approach debugging code including: (1) skills and competencies involved in debugging, (2) programmers' behavior patterns while debugging, and (3) factors that affect how well programmers debugs code.

2.3.1 Skills and competencies. A 2008 literature review [29] identified program comprehension and hypothesis generation as key skills during debugging.

Program comprehension, the ability to reason about what the code does, is unsurprisingly important in debugging code effectively [1, 20, 24, 33]. Ducasse and Emde [12] distinguish two types of reasoning about code behavior: "knowledge of the intended program" and "knowledge of the actual program". Here, the intended program represents the solution the programmer had in mind while writing the code. The actual program is what the code currently does. This distinction is particularly important when the person debugging the code is different from the person who wrote the code. Here, the author has no knowledge of the intended solution and must thus infer it based on the current code.

Effectively generating and testing hypotheses is also critical to debugging [17, 19, 44]. Studies have found that novices tend to hypothesize and test hypotheses more frequently, but have a lower rate of correct hypotheses than experts [19]. Tubaishat [43] distinguished between "deep reasoning" and "shallow reasoning" when forming hypotheses and finding bugs. Shallow reasoning involves using heuristics to make guesses about the output and test

results of the code, while deep reasoning relies specifically on deep program comprehension and recognition.

2.3.2 Attention patterns when debugging. Knowing how programmers allocate their attention while debugging code, and how attention patterns differ between expert and novice debuggers, is important for designing interfaces which help people debug more effectively. Several studies use eye tracking to analyze these attention patterns.

Unsurprisingly, the code being debugged consistently attracts the most attention [22, 42]. However, programmers with different expertise levels and different success levels at debugging seem to differ in how they trace through the code, and how they relate the code to other information, such as the output of the program and the debugger diagnostics. Lin et al.[27] found that less successful debuggers tend to trace through the code top-down rather than following the logical flow of the code. Bednarik[3] found that experts relied more on relating the code to the output of the program, while beginners paid more attention to visualizations of the program state which were displayed in the IDE side-by-side with the code.

Bednarik and Tukiainen[4] used a Restricted Focus Viewer(RFV) to blur most of the interface, giving the user control over which part to show using the mouse. They observed a "bookmarking" behavior in participants, where the participant is not looking at the unblurred portion of the screen, but rather is looking at a blurred portion while retaining another portion in the RFV. This can be interpreted as using the focused view as a form of external memory to temporarily "save" the information. They also found that the blurring slows down expert but not novice debuggers, in that experts fixate on individual elements of the UI for longer periods of time when debugging with the RFV. This indicates that experts take longer to process important information when they know that it will not be immediately available to them visually. Together, these findings suggest that visual cues in the interface can serve as an important resource for alleviating some of the high cognitive load associated with the debugging process.

2.3.3 Factors which affect debugging success. Several studies show that increased cognitive load negatively impacts novices' behavior when making and resolving bugs. McKeever and McDaid[30] demonstrated that students' ability to find certain kinds of bugs in spreadsheets decreases when the spreadsheets are formatted to provide additional information by giving meaningful names to ranges of cells and using these names in place of cell references. Anderson and Jeffries[2] found that artificially adding irrelevant complexity increases the frequency of errors that students make in Lisp programs, and that these errors tend to be slips rather than the result of misconceptions. They concluded that "errors occur when there is a loss of information in the working memory representation of the problem". Robertson et al.[41] investigated the difference between forcing users to react to debugging information immediately and making the same information available at the user's convenience. They found that interrupting users reduced their effectiveness in debugging, and attributed this to the way that frequent interruptions affect short-term memory. Wilcox et al.[48] examined the effect of continuous visual feedback on debugging, and found that it both helped and hindered, depending on the context.

Our results suggest that the availability of automatically corrected versions of students code help TAs to more readily reason about student intent and provide help that aligns with that intent, an important aspect of the debugging process [12]. We also found that TAs experienced short term memory lapses when providing feedback. Like programmers in Bednarik and Tukainen's study [4], TAs appeared to use student code as a form of external memory.

2.4 Automatic code correction

Researchers have explored the use of automatically generated hints to help struggling students. These systems generate corrections using a variety of strategies including mining solutions [18, 37, 40, 47, 49] or strategies [10, 21, 26, 34] from the work of other students, and relying on instructor written tests to check for correctness [23, 39, 47]. Most systems have then used their generated corrections to subsequently generate a student-facing hint that describes a step to take, but does not explicitly show the correction. However, at least one system instead applies this information to a teacher-facing interface [21] to reduce repetitive work associated with providing student feedback.

One study [38] evaluated the quality of next-step hints generated by several algorithms by comparing the generated suggestions with a corpus of "gold standard" edit suggestions generated by a group of experts. They found that the best algorithms performed between 47.9% and 83.9% as well as human tutors. The human tutors, in turn, performed between 75% and 90% as well as the gold standard. The study identified the ability to understand student intent as one of the major aspects that present difficulty for automatic code-fixing algorithms.

In this study, we found that poor automatic corrections slowed TAs down but did not significantly negatively impact the quality of their feedback.

3 METHODS

We ran a pair of studies to explore the potential impact of automatically generated code corrections on 1) TAs' identification of student errors, 2) TAs' debugging process, and 3) the quality of TAs' proposed solutions. In both studies, we asked TAs to evaluate broken student code submissions, fix them, and then provide written feedback to the student that would help them to move forward. We chose to have TAs provide written feedback in order to alleviate the time pressure that is inherent in live student interaction. This setting required TAs to fully debug student submissions before deciding on what feedback to offer. The first study focused on high quality generated code and explored code problems of varying complexity. The second study evaluated the potential impact of non-optimal generated code.

Specifically, we addressed the following research questions:

RQ1. Does seeing a comparison of student code with an automatically-generated corrected version of the same code help TAs debug the student code faster?

RQ2. Does seeing such a comparison help TAs debug the student code better (more accurately and completely)?

RQ3. If the generated corrected version is problematic, (e.g. suggests spurious changes or passes provided unit tests without being correct), how does it affect the TAs' ability to debug student code?

RQ4. Qualitatively, what presents the most difficulties for TAs when debugging student code? What can be done to help/support TAs as they debug student code?

3.1 Participants

We recruited study participants through a mailing list for current and future undergraduate computer science TAs at Washington University in St. Louis. Participants received a $15 gift card in recognition of their participation in an hour-long study session.

3.2 Study Materials

For this study, we created a TA web interface[1], selected eight real-world buggy student solutions to programming problems, and created two versions of corrections for each buggy student solution through a mix of automatic correction generation and manual adjustments to fit our specific study needs. We describe each below.

3.2.1 TA Web Interface. During the study, we asked participants to first fix incorrect student solutions and then provide feedback to students. To support this and capture information about their process, we created a simple web interface (see Figure 2).

At the top of the interface, we showed information about the problem: the problem statement that the student saw, the buggy student solution, and (in some conditions) the generated correction to the student solution. Participants used the lower half of the interface to debug the code and then write the explanation to the student. They could test their modified version of the code at any time, and the interface required all unit tests to pass before they could submit their answer.

3.2.2 Problem Data. To populate the TA web interface, we need the following content for each problem: a problem description, a buggy student solution to that problem, the unit tests used to evaluate a solution's correctness, and two different versions of a corrected solution to the student solution (described in more detail in the "Corrections" section below).

To provide a realistic set of problems and student submissions for TAs to debug and provide feedback on, we selected buggy solutions from a public dataset of real-world beginners solving Python problems[36]. We also used problem descriptions and unit tests from the same source. The dataset was designed to capture how novice programmers solved Python programming problems using the ITAP intelligent tutoring system[40]. The ITAP system provides on-demand hints to students when they are stuck and do not know how to correct their code. Thus, we expect that the kinds of errors that students may make while working with the ITAP system would be qualitatively similar to the kinds of errors a student might make while working on programming exercises with a TA on hand to assist them.

In order to mitigate the variance in difficulty inherent in debugging real-world beginner code, we hand-selected eight buggy student solutions to eight different programming problems and showed each participant the same eight problems. This way, all participants would be addressing the same bugs in the same context, and we could make consistent comparisons between different participants' performance. The number of problems was chosen so that most undergraduate TAs could feasibly complete all problems within the one-hour study.

We used several criteria to select the eight problems: (1) Half of the problems should have a small bug and half should have a larger issue. We defined solutions with large issues as ones involving errors in logical program flow and/or multiple interacting bugs; and solutions with small issues as not involving either of those. (2) Taken together, the buggy student solutions should represent a wide variety of types of bugs observed in the dataset, including typos, arithmetic/computation errors, and different types of logical errors. (3) For each buggy student solution, it should be feasible to come up with both a valid and a problematic, lower-quality version of a "generated" correction (see the section below for more details on the two different types of corrections).

Appendix A summarizes the selected student solutions, bugs present in each one, both versions of generated corrections, and other associated data.

3.2.3 Corrections. In order to answer the research questions we posed, we need to separately test the effect of high-quality and low-quality generated corrections: RQ3 examines the effect of low-quality corrections on TA behavior, while RQ1 and RQ2 explore the effect of seeing a correct and valid correction to the student code. Therefore, for each problem, we created both a valid correction which adequately addressed the bugs in the student code, and a problematic low-quality correction which, in some way, did not address all bugs, but nevertheless did pass all unit tests.

For each problem and buggy student solution, we used a three-step process to create both versions of the correction: (1) we automatically generated a correction using a custom algorithm; (2) we decided whether it was a valid or problematic correction; (3) finally, we manually modified the generated version to create the other version. These steps are described in more detail below.

For each problem, we first used a heuristic algorithm to automatically generate a candidate solution. This heuristic compares the buggy student solution to other students' correct solutions, which were taken from the same dataset as the buggy solution. Using the difference between the buggy and correct solutions, it tries to find a small subset of the edits which, when applied to the student code, allow it to pass all unit tests. This is a similar algorithm to the ones used in other correction generators described in the Automatic Code Correction sub-section of the Related Work section above, although we developed our own set of strategies and heuristics to compare the solutions and search for small sets of edits. The exact design of the system is out of scope for this paper, since the goal of this paper is not to evaluate a particular correction generation algorithm, but rather to examine the effect of generated corrections as a whole on TA behavior.

We then manually examined each generated solution to decide whether it was a valid correction for all the bugs present in the student code. In all cases but one, the latest iteration of our correction generator created a valid solution which addressed the individual bugs while respecting student intent. The exception was the student solution to the sumOfDigits problem, which uses a very esoteric approach to both iterate through the digits of a number and extract each individual digit. Since the generator could not match

[1]The code for the TA web interface is available at https://github.com/yanamal/TA-debug-interface

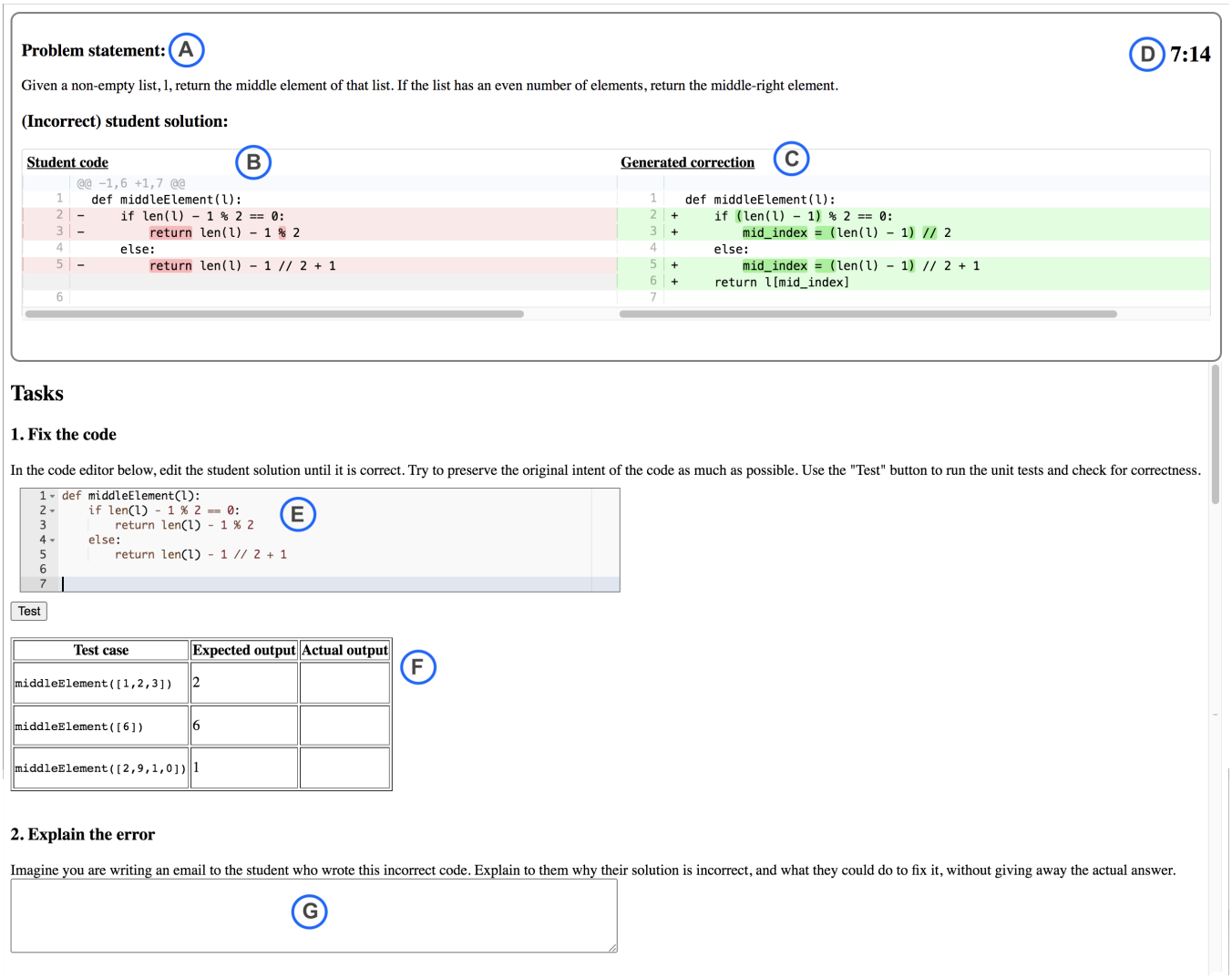

Figure 2: The debugging interface used in the study: (A) problem statement show to the student who wrote the buggy solution; (B) the student's solution, which is not correct (does not pass all unit tests); (C) A generated correction - shown in all conditions except the Control condition for the first experiment; (D) Countdown timer for a soft per-problem time limit; (E) Editable code area where the participant creates their own solution; (F) Unit tests which must pass before the participant can submit their answer; (G) text area where the participant writes the student-facing explanation of what is wrong with the student code.

this approach to any other student solutions, the automatically generated correction used a simplified and more common approach to solving the problem. But the student approach, despite being esoteric, would have worked if it weren't for four individual bugs. So for this study, we manually corrected each bug in the student code to create the valid "generated" solution version, and reserved the actual generated solution for the problematic solution. This allows us to study how teaching assistants react to esoteric student solutions in the presence or absence of information about what the student intended.

Finally, for each of the seven problems where the generated solution was valid, we created an alternative solution which was problematic in some way. This was either a solution generated by a previous version of the algorithm, or a manually-created problematic solution where we introduced a problem representative of ones that can result from code generation. Each problematic solution still passes all unit tests, since this is a requirement of the correction generator. The problems present in these problematic solutions vary from code that contains unnecessary changes to objectively incorrect code which exploits accidental loopholes in unit tests.

The resulting valid and problematic versions of the solution, as well as a description of what is wrong with each problematic version, are also shown in Appendix A.

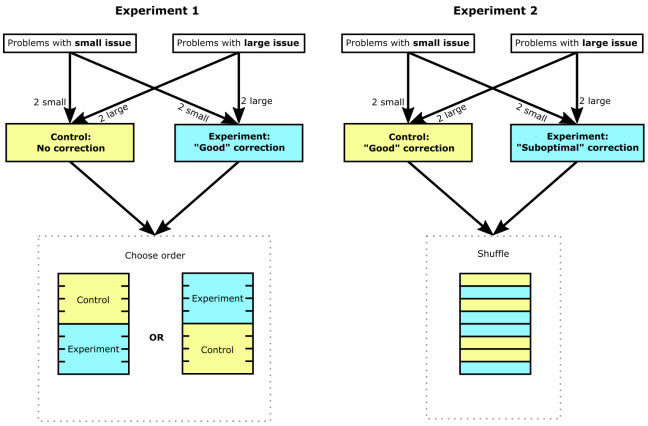

Figure 3: Overview of the experiment design

Table 1: Overview of how experiments relate to RQs

		Does experiment i address RQ j?			
Experiment	Participants	RQ1	RQ2	RQ3	RQ4
Experiment 1	17	✓	✓		✓
Experiment 2	5			✓	✓

3.3 Experiment Design

To answer the research questions, we conducted two separate within-subject experiments with different sets of participants. We chose a within-subject design because undergraduate teaching assistants have a very broad range of experience and ability in both teaching and programming. This can make it very hard to draw between-subject conclusions, since any variance induced by the controlled variable can be overshadowed by the natural variance in TA performance. Table 1 summarizes the relationship between experiments and research questions, and the number of participants in each experiment. Figure 3 shows an overview of the two experiments' designs.

3.3.1 Experiment 1. In the first experiment, we presented each participant with two versions of the interface. They first worked through four problems using one version of the interface, then switched to the other version for the other four problems. For problems in the experimental condition, the participant saw a comparison between the student code and the valid generated correction, with standard side-by-side diff highlighting. The control version did not show any correction or highlighting. For each participant, we randomly selected two problems with large issues and two problems with small issues to be in the experimental condition, with the other half of the problems in the control condition. We also randomized which condition the participant saw first, to control for any variance introduced by the learning curve of getting used to the interface.

3.3.2 Experiment 2. For the second experiment, participants saw a generated correction in both the experimental and the control condition. In the control condition, they saw the same valid correction as in the experimental condition from experiment 1. In the experimental condition, they saw the problematic correction described above. For each participant, the problems were split randomly between the two conditions the same way as in experiment 1: two small and two large problems in each condition. But unlike in experiment one, we randomly shuffled the resulting comparisons so that the participants would see a mix of the two conditions instead of switching between them midway through. The participants were not made aware that there were two conditions, and did not know whether the solution they were presented was valid.

3.4 Study Process

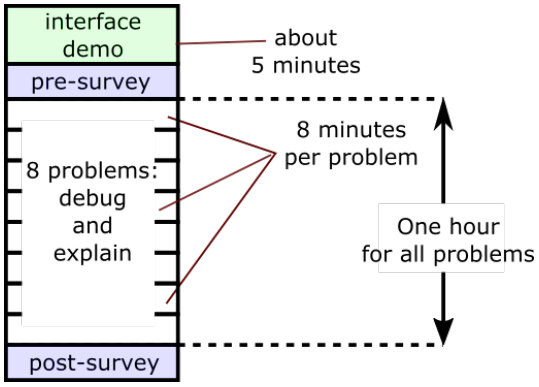

Figure 4: Overview of the study procedure

We conducted an individual one-on-one study session with each participant. At the beginning of the session, the researcher gave a 5-minute demonstration of the interface shown in Figure 2 and explained the participant's task, in order to minimize the time that participants spent familiarizing themselves with the interface.

After the demo, the participant filled out a short survey asking about their experience with Python and with being a Teaching Assistant, and then started working on the debugging tasks. For each debugging task, the participant was asked to first edit the student code until all tests pass, and then write an email to the student explaining the errors. If the participant took too long to fix the code for a particular problem, the requirement that all tests should pass was lifted, and the participant could choose to move on to the explanation at any time. We chose "too long" to be 8 minutes, so that the participants could still feasibly finish all 8 problems in around an hour. Although this meant that it was technically possible for a participant to take up to 64 minutes to debug all of the problems, we counted on most participants finishing at least a few problems before the deadline. This assumption turned out to be correct: most participants finished all 8 problems well within the 1-hour limit, even in cases where one or two problems took longer than 8 minutes.

The participant was not explicitly notified if the 1-hour limit passed while they were still working on the debugging tasks. But once they submitted the response to their current task, they were redirected to the post-survey stage, even if there were still problems left which they did not solve. This meant that two out of 22 participants did not complete all 8 problems. One of these participants was able to complete 7 problems in the hour-long study, while the other

only completed 6. All other participants completed all problems, and many finished early.

After each session, the participant filled out a short post-survey with two open-ended questions about what they found helpful and difficult in the process.

Figure 4 summarizes the study procedure described above.

3.5 Data Collection

We collected pre-survey and post-survey answers for each participant, as well as the logs from the debugging interface.

In the web interface, we recorded each time the participant ran a unit test. For each of these events, we logged the code that was executed, and the results of each unit test.

In addition, each time that the participant submitted an answer to a problem, we logged their code, the unit test results, and the full text of the explanation they wrote for the student.

3.6 Data Analysis

In order to evaluate the TAs' performance across different experimental conditions regardless of which problems they saw in which condition, we computed within-subject *relative performance metrics* which seek to control for the wide variance in expected performance between different problems.

Additionally, in order to evaluate the quality and accuracy of TAs' answers, we performed two analyses: (1) We developed criteria for evaluating whether the TA's version of the code *fixed* each bug present in the student solution, and (2) we coded each explanation to determine whether the TA's written explanation to the student *addressed* each of the bugs.

3.6.1 Relative Performance Metrics. To analyze within-subject variance across experimental conditions, for each metric of interest, we computed a problem-specific baseline across all participants. For each participant, we then measured the difference between that participant's performance on that metric and the problem-specific baseline. This strategy allowed us to capture, for each participant, their *relative* performance on each problem - whether they did better or worse than the baseline, and by how much. This allows us to control for the large amount of between-problem variance when performing a within-subject (paired) t-test to determine whether a participant's relative performance changed when the participant was subjected to different experimental conditions.

We performed this type analysis for two metrics: (1) the time to solve a particular problem, and (2) the size of the difference (edit distance) between the student's original code and the TA's corrected version.

Relative response time:

RQ1 directly concerns TA debugging time. To evaluate debugging time, we measured the time intervals between when the participant first saw the problem and when they submitted their student feedback. We calculated a problem-specific baseline by computing the average response time and standard deviation across all participants who worked on that problem. We define *relative response time* as the distance from the average response time measured in standard deviations for that problem. So, for example, a participant who took one standard deviation longer to complete a problem would have a relative response time of -1.

Relative edit distance:

To assess how well the participants addressed the student bugs, we wanted to know how well TAs preserved the student's original intent when editing their buggy code. To do this, we first calculated the edit distance between the original student code and the participant's corrected version. We define the edit distance as the number of fine-grained edits needed to change one version of the code into the other. In all of the selected student solutions, each bug could be corrected with a few edits, so smaller edit distances were more likely to preserve the students' original approaches. To calculate the *relative edit distance*, we normalized each TA's edit distance using the edit distance between the student's solution and the "valid" generated solution as the problem-specific baseline. Here, a positive relative edit distance means that the participant's correction made more changes to the student code than the generated correction and a negative relative edit signifies a solution closer to the student's intent

3.6.2 Evaluating Code Fixes. As part of answering RQ2, we evaluated whether each participant *fixed* each bug present in the student code. In order to capture the distinction between correcting bugs and working around them, we developed criteria which intentionally differentiate between fixing the specific expression that was incorrect in the student code, and coming up with some other solution to the original problem which worked around the bug in the student's buggy expression.

The criterion for deciding whether each bug is "fixed" in some modified version of the student code depended on whether the original bug was (1) an omission which could be corrected by inserting some code; or (2) an error which involved some incorrect expression in the student code, and this expression needed to be modified or deleted in order to fix the bug.

In cases where the bug could be corrected with inserting some specific code, e.g. a missing return condition, then the bug is considered "fixed" if the participant's correction inserted the necessary code. If the participant's correction never inserted the expected expression, the bug was considered "not fixed".

In cases where fixing the bug involved modifying some expression(s) in the student code, the criterion for "fixing" the bug hinged on the value of the expression in question. The bug was considered "fixed" if and only if the corresponding expression in the edited code evaluated to the same value as in the valid generated solution. On the other hand, if the participant's code completely removed the buggy expression, or left it to evaluate to an incorrect value, the specific bug was not considered "fixed" even if the participant's code correctly solved the original programming problem.

3.6.3 Coding of Explanations. We coded the participants' explanations to the hypothetical students to see whether they *addressed* each bug present in the student code. Since we asked the participants to "Explain to [the student] why their solution is incorrect, and what they could do to fix it", we could feasibly expect them to try to address each bug they were aware of. For each participant response and each bug present in the student solution to the relevant problem, we labeled the bug as either "addressed" or "not addressed" in the response.

The criteria for "addressing" a bug were that the TA made an attempt to explain or hint at the bug in the explanation, such that

if the student had understood the explanation and followed the advice, they would be able to fix the bug.

To test interrater reliability, two researchers coded all participant answers to the sumOfDigits problem. We chose to test interrater reliability on a single problem because switching between problems (and the bugs relevant to each one) would have presented a huge context switch to the rater, which could affect their accuracy and reliability. To ensure that the interrater sample was valid, we chose the problem which both had the most bugs and the most varied quality of participant responses (sumOfDigits). The raters achieved Fleiss' kappa of 0.852, which is generally interpreted to indicate "almost perfect agreement". After establishing this interrater reliability, one of the two researchers coded the rest of the data.

4 RESULTS

Table 2: t-test results for performance metrics in Experiment 1

Metric	t-statistic	Cohen's D	p-value
Relative response time	2.73	0.683	0.015
Unit test runs	3.32	0.83	0.0043
Relative edit distance	2.389	0.60	0.030

In this section, we report our findings in answer to each of our research questions: (RQ1) Does seeing a comparison of student code with an automatically-generated corrected version of the same code help TAs debug the student code *faster*? (RQ2) Does seeing such a comparison help TAs debug the student code *better*? (RQ3) What is the effect of seeing a *problematic* generated correction? (RQ4) What presents the most *difficulties* for TAs when debugging student code, and what can help alleviate these difficulties?

4.1 RQ1: Do generated corrections help TAs debug faster?

RQ1 asks whether seeing a generated solution helps TAs debug student code faster. To answer this, we used the relative response speed metric described in the Data Analysis section to compare the within-subject difference between the control and experimental conditions in Experiment 1. We found that seeing a comparison to a generated solution does help undergraduate TAs debug student solutions faster, both in terms of elapsed time and amount of trial-and-error involved. These results are summarized in Table 2.

We performed a paired t-test with the null hypothesis that the within-subject relative response speed would not be affected by whether the subject saw a comparison to a generated solution. On average, participants performed 0.56 standard deviations faster when they saw the generated solution. (t(17)=2.73, p=0.015, d=0.683 - moderate effect size). This corresponds to a 29% improvement in average per-problem response speed.

To evaluate the amount of trial-and-error involved in finding a solution, we performed a similar paired t-test to analyze whether seeing a generated solution affects the number of times the participant executes the unit tests while solving the problem. Again, for each participant and problem, we calculated the number of tests relative to the global average for that problem. We found that participants who saw a generated solution used 2.02 fewer unit tests, on average (t(17)=3.32, p=0.0043, d=0.83 - large effect size).

4.2 RQ2: Do generated corrections help TAs debug better?

RQ2 asks whether seeing a generated correction helps TAs debug code and write explanations that are better and more accurate than they would without the generated correction.

Although the quality of a response can be very subjective, and we can't know the effect that a particular TA's response would have on the student who had written the original buggy solutions, we can use several metrics to evaluate whether the TAs succeeded at finding and communicating the bugs in the student code.

Specifically, we compared the following measures of response quality across the two conditions in Experiment 1: (1) was the TA's corrected code close to the student's original solution? (2) was the TA's corrected code actually correct? (3) did the TA's free-text explanation to the student address all the bugs in the student's code?

These metrics are not meant to evaluate the pedagogical soundness of the explanations that the TAs provided. Rather, we explicitly limited the scope of our metrics to measure how thoroughly the TA understood and internalized the issues with the student code. This level of understanding defines a lower bound on how good the TA's explanation could possibly be. A TA cannot effectively address an issue which they do not understand, or which they could not remember at the time of discussing the code with the student.

We found that participant performance on all three measures improved when the participant was able to see a generated correction.

4.2.1 Edit Distance from Student Solution. As described in the Data Analysis section, we used the relative edit distance between the buggy student solution and the TA's corrected solution as a proxy for whether the TA's corrected code preserved the student's original intent. The eight sample student solutions that were chosen for this study each had discrete bugs which could be corrected by a few edits. So, if a correction successfully preserved student intent, it would correct each of these bugs with few edits. On the other hand, corrections which are vastly different from the original code, and require many edits, are very unlikely to have preserved the student intent - they are much more likely to be rewrites that change the way that the code solves the problem.

We found that seeing a comparison with a generated solution enables TAs to come up with a smaller, more concise correction of their own.

In 6 of the 133 problems in this data set, the participant did not come up with a solution which passed all unit tests, because they ran out of time. We omitted these 6 problems from this particular analysis, since an incomplete solution could be arbitrarily close to the student code.

We performed a within-subject t-test of relative improvement in edit distance across conditions. We found that having a generated solution available allowed participants to shorten their relative edit distance by 5.70 edits on average (t(17)=2.389, p=0.030, d=0.60 - moderate effect size).

Notably, even though the generated solutions were specifically optimizing for a short edit distance, the participants in the study often found solutions with shorter edit distances than the generated solution. This indicates that the improvement in edit distance doesn't just come from copying the exact edits in the generated solution - rather, participants derived information from the generated solution which then helped them come up with short, concise corrections.

4.2.2 Correctness of Solution. As mentioned above, in 6 out of 133 cases, the participant never came up with a solution which passed all unit tests. In addition, 7 of the solutions that did pass all provided unit tests had uncaught bugs in them which would have been exposed by more comprehensive set of unit tests. For example, in the howManyEggCartons problem, three of the participants saw that only one unit test was failing, and chose to patch the code by introducing a special case for just that one unit test. If the problem had more unit tests where the number of eggs was even but not divisible by 12, these solutions would not work.

We performed a two-tailed Fisher's exact test to see whether seeing a generated solution affected the odds of coming up with a correct solution. We found that participants were more likely to submit an incorrect solution in the control condition of Experiment 1 (when they did not see a generated correction). The odds ratio for submitting an incorrect solution was 3.940 (p=0.042). So, the participants' odds of submitting an incorrect solution were nearly 4 times bigger in the control condition. This is usually interpreted as a medium effect size, though guidelines for interpreting odds ratios vary from source to source [8, 9].

4.2.3 Bugs Fixed and Addressed. Finally, we examined the data from the participants' code corrections and free-text explanations to see whether seeing the generated correction helped participants fix and address more of the bugs present in the student code.

Using all bugs present in the student code, we performed a two-tailed Fisher's exact test on both the bug-fixing and bug-addressing data we describe in the Data Analysis section.

Table 4 shows a summary of the results. We found that participants did have a much easier time both fixing the bugs in their code, and subsequently addressing them in the explanation to the student, when provided a code correction. The odds ratio for being able to fix a bug at any point in the debugging process was 17.762 (p=1.583e-12). So, the odds of fixing a particular bug were nearly 18 times bigger when participants saw the code correction. This is considered a very large effect size.

The odds ratio for addressing the bugs (regardless of whether they were fixed) is 2.940 (p=0.000248), which is a less drastic difference, and is generally interpreted as a small-to-medium effect size. Nevertheless, it is well within the bounds for being statistically significant.

We can see from the contingency data in Table 3 that participants addressed 65% of bugs in the control condition. They were able to address 84% of bugs in the experimental condition, or 1.30 times the control condition; so the *probability* of addressing a given bug improved by about 30% across conditions.

It is notable that participants often failed to address a bug in their explanations even if they did fix it in the code. This interesting result is explored further in the RQ4 results section below, which provides the qualitative analysis of what is difficult for TAs.

4.3 RQ3: What is the effect of seeing a problematic generated correction?

In RQ3, we consider the effect of seeing a problematic solution on TA performance. Any system for automatically generating fixes to student code will necessarily use some heuristics to guess at the fixes, and these heuristics will sometimes be wrong. In fact, as we described in the Related Work section, the state-of-the-art hint generation algorithms tend to perform with around 50-85% accuracy. But to what extent are these wrong generated solutions harmful? Can the potential for incorrect generated solutions outweigh any benefit of generating the solutions at all?

To test this, we analyzed data from Experiment 2, which compared showing the effect of a valid generated correction to the effect of showing a problematic correction. We performed the same analyses as we did for Experiment 1 when answering RQ1 and RQ2.

Participants performed 0.61 standard deviations slower with a problematic correction (t(5)=-4.740, p=0.009, d=-2.370) and used 2.25 more unit tests (t(5)=-2.835, p=0.047, d=-1.418). Relative edit distance increased by 0.25 edits when participants were shown a problematic correction (t=-0.238, p=0.824). For fixing and addressing bugs, the odds ratios were 0.3 (p=0.0711) and 1.286 (p=0.777), respectively. These statistics are summarized in Tables 6, 5, and 7.

We can conclude that seeing a problematic generated solution does slow TAs down when debugging code. In fact, the effect of seeing a problematic solution on response speed seems to be stronger than not seeing any solution at all - as discussed in section 4.1, the difference in speed between not seeing a solution and seeing valid generated solution was 0.56 standard deviations, which is a smaller difference than the difference observed in this experiment between participants seeing a valid or a problematic generated correction (0.61 standard deviations). So, it may take longer for a TA to understand an imperfect suggestion than it would if they had no solution to look at at all.

On the other hand, we cannot conclude that introducing problems to a generated solution affects the quality of the TA's own solution - in fact, although the results above which concern addressing bugs show no statistical significance, they lean toward supporting the null hypothesis that seeing problematic solutions does not have an effect on the TA's ability to provide explanations which address student bugs.

One interpretation is that although a problematic solution does slow the TA down and make them fall back on manually debugging the code, some parts of the generated solution are still informative, and still give the TA useful and usable ideas about what is wrong with the student code, and how to address the issues while preserving the original intent.

4.4 RQ4: What is difficult for TAs, and what is helpful?

RQ4 asks what presents the most difficulties for TAs, and what can help alleviate these difficulties.

Given the small number of participants we were able to recruit for the study, it is difficult to break down the data further for statistical

Table 3: Contingency tables for binary metrics in Experiment 1

Condition	Solution		Bugs		Bugs	
	correct	incorrect	fixed	unfixed	addressed	unaddressed
Control (no correction)	55	10	86	47	86	47
Experiment (valid correction)	65	3	130	4	113	21

Table 4: Fisher's exact test results for binary metrics in Experiment 1

Metric	odds ratio	p-value
Correctness of solution	3.940	0.042
Bugs fixed	17.762	1.583e-12
Bugs addressed	2.940	0.000248

analysis based on individual problems and bugs. However, the data still suggest certain patterns in what aspects of the task were hard or easy for participants.

To address what aspects of the task seem hard for TAs, we first observe two particular aspects that do not seem to present many difficulties to participants. Knowing what types of things tend to be easy allows us to narrow down the reasons other tasks might be hard.

We then examine those bugs in the student code which proved the hardest to fix. For the three hardest bugs present in the study, only half of the participants (11 out of 22) were able to find and fix each one. Each of these bugs suggests a specific source of difficulty for the TAs: understanding student intent; finding and fixing the same bug in multiple places; and noticing and addressing bugs that are visually very small (the change affects a single operator in an arithmetic expression).

Finally, we discuss what interventions and interface features might be helpful in alleviating the difficulties that TAs have when finding and explaining issues in student code.

4.4.1 What is not difficult? Two particular aspects of debugging did not seem to present many difficulties to the participants, in that most participants were able to perform these tasks correctly: (1) creating a correct solution to the original programming problem and (2) addressing bugs that do not interact with other bugs.

As we described in the results section for RQ2, the participants in Experiment 1 wrote correct solutions for 120 of 133 problems they were solving. Additionally, in Experiment 2, all participants wrote correct solutions for all problems they attempted. The fact that the majority of the participant-written solutions were correct suggests that at least for introductory-level problems, the difficulties that TAs encounter are not, in general, due to misconceptions or lack of understanding about the programming task itself. These participants are able to solve these types of programming problems consistently, under a short time limit, and while devoting a large part of their attention to other tasks involved in debugging and communicating with students.

In addition, all participants were extremely successful at addressing the bugs in 5 of the 8 programming problems. For these five problems, at most two participants failed to address each bug present in the problem (see Table 8). All other TAs addressed all bugs in their explanations. These problems covered a wide variety of types of bugs: arithmetic errors, logical flow errors, forgetting to fulfill part of the problem statement. Two of these problems (anyLowercase and nearestBusStop) were labeled as having "large" issues in the student code, because fixing these issues necessitated significant changes to the logic of the solution.

The common theme among the remaining 3 problems (howManyEggCartons, middleElement, and sumOfDigits) was that they involved bugs which interacted with each other in how they affected the output.

This suggests that participants were familiar with, and capable of dealing with, all kinds of bugs that are common to beginner programming code. The task of finding and explaining bugs only starts presenting difficulties when the participant is overwhelmed with the complexity that comes with interacting bugs.

4.4.2 Understanding Student Intent. In a post-survey, we asked participants: "What was the hardest or most cumbersome part of the task?" 11 out of 22 participants explicitly described understanding and adapting to student intent as the hardest part. For example, one participant described the hardest part of the task as: "Reviewing the code and understanding the student's logic. The student's solution sometimes does not approach the problem the same way I would, so it takes me a moment to understand what they're trying to do."

There is also some support for this in the data: Two of the three hardest-to-fix bugs appear in the sumOfDigits problem, which is the only problem with a manually-written "valid" solution. As described in the Study Materials section, this was necessary because the student's solution was extremely esoteric, and this was the only case where the generated solution failed to preserve student intent.

One of these two bugs directly affected part of a calculation that was very hard to interpret with that bug in place (the other one is discussed below, in the section about single-character bugs). This calculation was trying to use integer division by a power of 10 in order to "cut off" the number at a certain digit (see Appendix A.8 for details, including the buggy and corrected versions of the code). However, as is, the calculation almost always resulted in 0, because the power of 10 was too big. If, instead of n, the student had used the number of digits of n, the calculation would have worked correctly, making the subsequent calculations isolate the ith digit of n. Knowing that, it seems likely that this is what the student intended. But it is really hard to infer that intent from the calculation itself.

The manually-written "valid" correction substitutes `len(str(n))` in place of `n` on that line of code, which makes that line of code work correctly. Only one participant who saw the "valid" correction did not fix this bug; and conversely, only one person who did not

Table 5: Contingency tables for binary metrics in Experiment 2

Condition	Bugs fixed	Bugs unfixed	Bugs addressed	Bugs unaddressed
Control (valid correction)	40	4	35	9
Experiment (problematic correction)	27	9	30	6

Table 6: t-test results for performance metrics in Experiment 2

Metric	t-statistic	Cohen's D	p-value
Relative response time	-4.740	-2.370	0.009
Unit test runs	-2.835	-1.418	0.047
Relative edit distance	-0.238	-0.119	0.834

Table 7: Fisher's exact test results for binary metrics in Experiment 2

Metric	odds ratio	p-value
Bugs fixed	0.3	0.0711
Bugs addressed	1.286	0.777

see the valid correction did fix this bug. Of all the people who did not fix this bug (according to the criteria for "fixing a bug" outlined in the data analysis section), only two actually failed to come up with a correct solution to the original programming problem. This means that 9 of the participants worked around this bug, usually by completely rewriting the calculation for the ith digit. Thus, most people who saw the suggested change accepted it as the correct thing to do; but most people who did not see the change could not figure out what the student intended with that line of code.

On the other hand, only 6 of the 11 people who fixed the bug went on to address it in their explanation to the student. The other 5 omitted that bug from their explanation, and talked about the other three bugs present in the student's code. This may indicate that even seeing a plausible correction to an esoteric calculation does not always allow TAs to understand the reason for the error and the correction well enough to be able to talk about it.

The participants' behavior with respect to this bug highlights both the difficulty and the importance of understanding the student's original intent when writing the code: when the intent is non-standard, it can be difficult for TAs to understand it well enough to identify individual bugs in the code. And when the TAs do not understand the student's intent, this affects their ability to address and explain the problem with the student code.

4.4.3 Finding and addressing single-operator bugs. The second of the three hardest-to-fix bugs also appeared in the sumOfDigits problem, and involved a single-character fix to a comparison operator: the student's code compared two values with <= in the loop's condition, but this created an unexpected extra iteration through the loop. The simplest fix is to use < instead. This bug proved to be both hard to fix and hard to address even when fixed. 6 of the 11 participants who fixed this bug never addressed it in their explanation to the student.

In fact, the only bug that was fixed but unaddressed more often was also a bug with a single-operator fix: in middleElement, a calculation used % instead of //. This bug was fixed but left unaddressed by 8 participants in the study.

Both of these bugs are visually very small (involving one or two characters), and share the line of code with another, more prominent bug. But qualitatively, these two bugs are quite different: one is a logical error which affects the overall control flow of the program, while the other one is likely a result of a slip or typo. Yet the participants' behavior with respect to these bugs is quite similar: the participants tended to have a lot of difficulty finding each bug; and when they did find it, they were very likely to neglect to address it in their explanation.

This is somewhat counter-intuitive: if the bug was difficult to find, we might expect the participant to consider it important to alert the student to it. One plausible explanation is that participants across conditions rely on visual cues in the interface (the code and, when present, the diff) as a form of external memory to track bugs. Since these bugs are visually small and hard to notice, they are more likely to be missed both when looking for bugs and when trying to recall what needs to be explained to the student.

This behavior highlights a deficiency of a simple diff view between two versions of the code: the two bugs were highlighted in the diff, but they were small enough and near enough to other bugs that the participants could not effectively keep track of them as separate entities, even if they did find them. If the interface instead somehow explicitly identified these changes as separate bugs, it would be harder for the participants to forget about them when composing an explanation to the student.

4.4.4 Fixing multiple instances of the same bug. The final of the three hardest-to-fix bugs occurred in the middleElement problem. The student solution for this problem omits parentheses in 3 near-identical cases, when calculating `(len(l) - 1)//2` or `(len(l) - 1)%2`. When debugging the student code, 19 out of the 22 total participants were able to find and fix at least one of these cases. However, only 11 of those participants ended up finding all 3 of the cases.

The other participants would usually modify the code until the parentheses were no longer necessary, by changing the expressions to not use a -1 (except one participant who never finished debugging the code). This means that their solutions essentially masked the parenthesis bugs instead of resolving them, which is suboptimal from the point of view of knowing what is wrong with the student code.

Moreover, out of the 11 participants who did fix all the parenthesis issues, only 4 found and fixed all 3 instances in one attempt,

Table 8: Per-bug statistics: for each bug, how many total participants (a) fixed that bug in the code; (b) addressed it in the explanation; (c) fixed in code, but did not address in the explanation; (d) did not fix in code, but did address it in the explanation. See appendix A for detailed description of each bug.

Problem	Bug	Fixed	Addressed	Fixed but unaddressed	Unfixed but addressed
hasTwoDigits	upper bound	22	22	0	0
	returns None	22	21	1	0
secondHalf	off by one	21	20	1	0
listOfLists	not sorted	18	20	0	2
howManyEggCartons	uses quad	16	15	2	1
	uses %2	15	13	4	2
nearestBusStop	rounding down	20	21	0	1
	rounding up	20	21	0	1
anyLowercase	all lowercase	22	21	1	0
middleElement	parentheses	11	13	3	5
	% instead of //	19	11	8	0
	returns index	21	18	3	0
sumOfDigits	nsize	17	18	2	3
	r calculation	11	8	5	2
	infinite loop	17	17	2	2
	extra iteration	11	5	6	0

without testing intermediate states with partial fixes. For the 7 who fixed the bug in stages, it often took fewer tries to find, understand, and fix the first instance than to fix the subsequent instances. On average, it took these participants 2.18 unit test runs to fix the first instance of missing parentheses, but 3.43 additional unit test runs to find all subsequent near-identical instances.

This indicates that in general, the participants were capable of conceptualizing and noticing the error. However, it was much harder for participants to notice and fix subsequent instances of an error they already found. Many participants either failed to find the additional instances altogether, or took longer to find additional instances than the first instance. When participants could not track down the additional instances of the error, they compensated by making larger-than-necessary changes to the student code, thus masking the error.

4.4.5 What is helpful? In the post-survey, we also asked participants: "What was the most helpful part of the interface as you were trying to identify the problems with the student code?"

In answer to this question, 15 participants mentioned the generated correction. For example, one participant said: "The suggested solutions were helpful. They made it easier to tell what the program was generally looking for in terms of correction, **though not always the exact change**. " (emphasis added). Additionally, 11 participants mentioned the clear and concise presentation of unit tests (many participants chose to talk about several most helpful part of the interface). Another participant said the following about the unit tests: "The testing button was simple and intuitive, and I liked how clearly it outlined the cases that succeeded and failed. **It would be nice if all testing was that easy**, as a student and as a teacher." (again, emphasis added)

The fact that so many participants explicitly mentioned the unit tests is particularly interesting, because this aspect of the interface was not part of the intended study - it was just a simple design chosen to facilitate using the rest of the interface.

The results for RQ1 and RQ2 described above support the idea that the generated correction was helpful to TAs. Though we did not test for this explicitly, the data also support the idea that the unit test interface facilitated the TAs' task.

In 81 of the 173 total problems solved as part of the study, the participant continued changing and testing code after they arrived at a version of the code which passed all unit tests. They changed the code from a correct state to be more incorrect, even though they knew they would need to fix the code again before submitting their response. Examining the available video logs shows that participants did this while writing explanations to the students, in order to ensure that they are not making any mistakes in their explanations, and sometimes in order to provide concrete examples of buggy code behavior to the students.

The participants knew exactly what to test for in order to bring up the information they needed, but they relied on the unit testing interface to be able to recall the specifics of the information. Arguably, they were relying on the simple and fast unit testing interface to act as a source of external memory, so that they could offload some of the contents of their working memory onto the interface.

5 DISCUSSION

In general, undergraduate TAs who participated in the study do not tend to struggle with the conceptual task of identifying and reasoning about individual bugs present in student code. The major difficulties seem to come from high cognitive load that is extrinsic to code comprehension and "deep reasoning" about the code itself.

In particular, participants tended to do worse when there were several bugs that interacted with each other; when they needed to recognize and address additional instances of a bug they already

saw and understood; and when they needed to remember a bug that was difficult to fix, but also difficult to see (because it only took up one or two characters).

At the same time, the most helpful interface elements were those that alleviated some of this extrinsic cognitive load, by giving participants visual representations of the information they needed to complete their tasks: a summary of the bugs they were trying to address, in the case of the comparison to the generated code correction; and a concise, readable, and easy-to-access set of examples for the consequences of the students' bugs, in the case of the unit tests.

The other major source of difficulty for participants was inferring student intent from the code. This was both explicitly identified by the participants themselves in the post-survey, and seen in the participants' behavior when they tried to deal with an esoteric and hard-to-parse student solution.

The fact that the participants emphasized the difficulty of inferring student intent indicates that they were aware the importance of understanding student intent, and self-reflected on their ability to do so. But although the participants did try to respect the student's original intent while debugging the code, they tended to fall back on rewriting the solution when they ran into roadblocks because of the extrinsic difficulties mentioned above.

This highlights the importance of mitigating these types of difficulties in order to help TAs provide more effective assistance to students. And although this study focused specifically on undergraduate TAs, these findings can generalize to anybody trying to help a student or beginner programmer with their code. Undergraduate teaching assistants are not the only people who encounter the kinds of difficulties described above. The frustration of spending far too long looking for a really simple bug, and the temptation to blow everything away and start over, are familiar to most people who have engaged in debugging code, especially code written by someone else. Although the impact of these types of difficulties may be stronger for less experienced programmers like undergraduate TAs, understanding and alleviating them could make the job of helping students easier for everyone.

6 LIMITATIONS AND FUTURE WORK

The biggest limitation of this study is the small sample size of participants. Although this set of data allowed us to draw statistical conclusions about the overall effect of seeing different types of code corrections, in the future, we would like to conduct similar studies at a larger scale. This would allow us to analyze more rigorously how TAs approach debugging different types of problems, what difficulties they encounter, and how a generated correction can help.

There are also some limitations in how realistically the experiment design represented a TA's real-world task of helping a student. Although we used real-world student code, we presented the participants with a task that doesn't completely capture the teaching practices a TA would need to engage in in order to effectively help a student. The participants were not required to provide live assistance to the student, and were not evaluated on whether the explanations they provided constitute effective teaching of the underlying concepts. On the other hand, they *were* asked to completely debug the code and also completely explain all the bugs in their written explanation. In a real-world situation, a TA may choose to focus on just some of the issues in their discussion with the student, and give the student an opportunity to resolve the other issues themselves. Further, providing a direct explanation of a bug - as we asked the participants to do in this study - may not be the most effective way to help the student learn how to find or avoid this type of bug in the future. We made these design choices in order to maximize the amount of information we could get about what the TAs are *capable of understanding and articulating* in 8 minutes of interacting with student code. But in future studies, we plan on testing similar systems in more realistic environments.

Finally, as discussed in the section about single-operator bugs, there seems to be a limitation in how useful a simple diff view is to a TA debugging student code. Therefore, we are currently working on a version of the correction-generating algorithm and interface which would break up the difference into individual bug fixes, and provide on-demand detailed information about how each bug fix improves the outcome of running code.

7 CONCLUSION

This work contributes an understanding of how users interact with code, specifically, how TAs in undergraduate computer science courses interact with broken student code when they are tasked with debugging it. It evaluates the effect of seeing a corrected version of the student code on this interaction.

We showed that seeing a valid correction allows the TA to both debug faster and write better student-facing explanations of their bugs. We also showed that seeing a *problematic* correction slows TAs down in comparison with seeing a valid correction. However, we were not able to draw statistical conclusions about whether TAs provide worse explanations when given a problematic correction.

Finally, we presented evidence that high cognitive load plays a large role in TAs making mistakes and generally having difficulty when helping a student with their code. We argued that tools which help mitigate this cognitive load may help TAs be more effective in helping students.

These findings can help inform the design of TA-facing interfaces that assist TAs in understanding and addressing student issues in CS courses.

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '05)*. Association for Computing Machinery, Caparica, Portugal, 84–88. https://doi.org/10.1145/1067445.1067472

[2] John R. Anderson and Robin Jeffries. 1985. Novice LISP Errors: Undetected Losses of Information from Working Memory. *Human–Computer Interaction* 1, 2 (June 1985), 107–131. https://doi.org/10.1207/s15327051hci0102_2 Publisher: Taylor & Francis _eprint: https://doi.org/10.1207/s15327051hci0102_2.

[3] Roman Bednarik. 2012. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International Journal of Human-Computer Studies* 70, 2 (2012), 143–155. Publisher: Elsevier.

[4] Roman Bednarik and Markku Tukiainen. 2005. Effects of display blurring on the behavior of novices and experts during program debugging. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems*. ACM, Portland OR USA, 1204–1207. https://doi.org/10.1145/1056808.1056877

[5] Maureen Biggers, Tuba Yilmaz, and Monica Sweat. 2009. Using collaborative, modified peer led team learning to improve student success and retention in intro cs. In *Proceedings of the 40th ACM technical symposium on Computer science education*. 9–13.

[6] Rebecca Brent, Jason Maners, Dianne Raubenheimer, and Amy Craig. 2007. Preparing undergraduates to teach computer applications to engineering freshmen. In *2007 37th Annual Frontiers In Education Conference-Global Engineering: Knowledge Without Borders, Opportunities Without Passports*. IEEE, F1J–19.

[7] Mark J. Canup and Russell L. Shackelford. 1998. Using software to solve problems in large computing courses. *ACM SIGCSE Bulletin* 30, 1 (1998), 135–139. Publisher: ACM New York, NY, USA.

[8] Henian Chen, Patricia Cohen, and Sophie Chen. 2010. How Big is a Big Odds Ratio? Interpreting the Magnitudes of Odds Ratios in Epidemiological Studies. *Communications in Statistics - Simulation and Computation* 39, 4 (March 2010), 860–864. https://doi.org/10.1080/03610911003650383

[9] Susan Chinn. 2000. A simple method for converting an odds ratio to effect size for use in meta-analysis. *Statistics in Medicine* 19, 22 (2000), 3127–3131. https://doi.org/10.1002/1097-0258(20001130)19:22<3127::AID-SIM784>3.0.CO;2-M _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1097-0258%2820001130%2919%3A22%3C3127%3A%3AAID-SIM784%3E3.0.CO%3B2-M.

[10] Sammi Chow, Kalina Yacef, Irena Koprinska, and James Curran. 2017. Automated data-driven hints for computer programming students. In *Adjunct Publication of the 25th Conference on User Modeling, Adaptation and Personalization*. 5–10.

[11] Adrienne Decker, Phil Ventura, and Christopher Egert. 2006. Through the looking glass: reflections on using undergraduate teaching assistants in CS1. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. 46–50.

[12] M. Ducasse and A.-M. Emde. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings. [1989] 11th International Conference on Software Engineering*. 162–171. https://doi.org/10.1109/ICSE.1988.93698

[13] Ronald Erdei, John A. Springer, and David M. Whittinghill. 2017. An impact comparison of two instructional scaffolding strategies employed in our programming laboratories: Employment of a supplemental teaching assistant versus employment of the pair programming methodology. In *2017 IEEE Frontiers in Education Conference (FIE)*. 1–6. https://doi.org/10.1109/FIE.2017.8190650

[14] Joseph Fong, Dawn Leung, and Donny Lai. 2009. A Peer-to-Peer eLearning Supporting System for Computer Programming Debugging System. In *Hybrid Learning and Education*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Fu Lee Wang, Joseph Fong, Liming Zhang, and Victor S. K. Lee (Eds.). Vol. 5685. Springer Berlin Heidelberg, Berlin, Heidelberg, 230–239. https://doi.org/10.1007/978-3-642-03697-2_22 Series Title: Lecture Notes in Computer Science.

[15] Meg Fryling, MaryAnne Egan, Robin Y. Flatland, Scott Vandenberg, and Sharon Small. 2018. Catch'em Early: Internship and Assistantship CS Mentoring Programs for Underclassmen. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 658–663.

[16] Elena L. Glassman, Christopher J. Terman, and Robert C. Miller. 2015. Learner-Sourcing in an Engineering Class at Scale. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale (L@S '15)*. Association for Computing Machinery, New York, NY, USA, 363–366. https://doi.org/10.1145/2724660.2728694

[17] John D. Gould and Paul Drongowski. 1974. An exploratory study of computer program debugging. *Human Factors* 16, 3 (1974), 258–277. Publisher: SAGE Publications Sage CA: Los Angeles, CA.

[18] Sebastian Gross, Bassam Mokbel, Benjamin Paaßen, Barbara Hammer, and Niels Pinkwart. 2014. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 10* 9, 3 (2014), 248–280. Publisher: Inderscience Publishers Ltd.

[19] L. Gugerty and G. Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*. Association for Computing Machinery, Boston, Massachusetts, USA, 171–174. https://doi.org/10.1145/22627.22367

[20] Leo Gugerty and Gary M. Olson. 1986. Comprehension differences in debugging by skilled and novice programmers. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. Ablex Publishing Corp., Washington, D.C., USA, 13–27.

[21] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale - L@S '17*. ACM Press, Cambridge, Massachusetts, USA, 89–98. https://doi.org/10.1145/3051457.3051467

[22] Prateek Hejmady and N. Hari Narayanan. 2012. Visual attention patterns during program debugging with an IDE. In *Proceedings of the Symposium on Eye Tracking Research and Applications - ETRA '12*. ACM Press, Santa Barbara, California, 197. https://doi.org/10.1145/2168556.2168592

[23] Wei Jin, Tiffany Barnes, John Stamper, Michael John Eagle, Matthew W. Johnson, and Lorrie Lehmann. 2012. Program representation for automatic hint generation for a data-driven novice programming tutor. In *International conference on intelligent tutoring systems*. Springer, 304–309.

[24] Irvin R. Katz and John R. Anderson. 1987. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399. Publisher: Taylor & Francis.

[25] Brittany Ann Kos. 2017. The collaborative learning framework: Scaffolding for untrained peer-to-peer collaboration. (2017).

[26] Timotej Lazar, Aleksander Sadikov, and Ivan Bratko. 2017. Rewrite Rules for Debugging Student Programs in Programming Tutors. *IEEE Transactions on Learning Technologies* 11, 4 (2017), 429–440.

[27] Yu-Tzu Lin, Cheng-Chih Wu, Ting-Yun Hou, Yu-Chih Lin, Fang-Ying Yang, and Chia-Hu Chang. 2016. Tracking Students' Cognitive Processes During Program Debugging—An Eye-Movement Approach. *IEEE Transactions on Education* 59, 3 (Aug. 2016), 175–186. https://doi.org/10.1109/TE.2015.2487341 Conference Name: IEEE Transactions on Education.

[28] Yi Liu, Gita PhelpsA, and Fengxia Yan. 2019. Developing a guided peer-assisted learning community for CS students. *Journal of Computing Sciences in Colleges* 34, 7 (2019), 72–80. Publisher: Consortium for Computing Sciences in Colleges.

[29] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92. Publisher: Taylor & Francis.

[30] Ruth McKeever and Kevin McDaid. 2010. How do Range Names Hinder Novice Spreadsheet Debugging Performance? *arXiv:1009.2765 [cs]* (Sept. 2010). http://arxiv.org/abs/1009.2765 arXiv: 1009.2765.

[31] Mia Minnes, Christine Alvarado, and Leo Porter. 2018. Lightweight Techniques to Support Students in Large Classes. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, Baltimore Maryland USA, 122–127. https://doi.org/10.1145/3159450.3159601

[32] Diba Mirza, Phillip T. Conrad, Christian Lloyd, Ziad Matni, and Arthur Gatin. 2019. Undergraduate Teaching Assistants in Computer Science: A Systematic Literature Review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3291279.3339422

[33] Murthi Nanja and Curtis R. Cook. 1987. An analysis of the on-line debugging process. In *Empirical studies of programmers: Second workshop*. 172–184.

[34] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2018. The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *arXiv:1708.06564 [cs]* (June 2018). http://arxiv.org/abs/1708.06564 arXiv: 1708.06564.

[35] Inna Pivkina. 2016. Peer learning assistants in undergraduate computer science courses. In *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–4.

[36] Thomas Price. 2021. thomaswp/CSEDM2019-Data-Challenge. https://github.com/thomaswp/CSEDM2019-Data-Challenge original-date: 2018-12-30T21:05:59Z.

[37] Thomas Price, Rui Zhi, and Tiffany Barnes. 2017. Evaluation of a Data-Driven Feedback Algorithm for Open-Ended Programming. *International Educational Data Mining Society* (2017).

[38] Thomas W. Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A Comparison of the Quality of Data-driven Programming Hint Generation Algorithms. *International Journal of Artificial Intelligence in Education* 29, 3 (2019), 368–395. Publisher: Springer.

[39] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. 39–40.

[40] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (March 2017), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[41] T. J. Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R. Ruthruff, Laura Beckwith, and Amit Phalgune. 2004. Impact of interruption style on end-user debugging. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 287–294.

[42] Pablo Romero, Richard Cox, Benedict du Boulay, and Rudi Lutz. 2002. Visual Attention and Representation Switching During Java Program Debugging: A Study Using the Restricted Focus Viewer. In *Diagrammatic Representation and Inference (Lecture Notes in Computer Science)*, Mary Hegarty, Bernd Meyer, and N. Hari Narayanan (Eds.). Springer, Berlin, Heidelberg, 221–235. https://doi.org/10.1007/3-540-46037-3_23

[43] Abdallah Tubaishat. 2001. A knowledge base for program debugging. In *Proceedings ACS/IEEE International Conference on Computer Systems and Applications*. IEEE, 321–327.

[44] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494. Publisher: Elsevier.

[45] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Jaakko Kurhila. 2013. Massive increase in eager TAs: Experiences from extreme apprenticeship-based CS1. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 123–128.

[46] Erin Walker, Nikol Rummel, and Kenneth R. Koedinger. 2009. Integrating collaboration and intelligent tutoring data in the evaluation of a reciprocal peer tutoring environment. *Research and Practice in Technology Enhanced Learning* 4, 03 (2009), 221–251. Publisher: World Scientific.

[47] Ke Wang, Benjamin Lin, Bjorn Rettig, Paul Pardi, and Rishabh Singh. 2017. Data-driven feedback generator for online programing courses. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 257–260.

[48] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems?. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. ACM, Atlanta Georgia USA, 258–265. https://doi.org/10.1145/258549.258721

[49] Kurtis Zimmerman and Chandan R. Rupakheti. 2015. An automated framework for recommending program elements to novices (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 283–288.

A PROGRAMMING PROBLEMS AND STUDENT SOLUTIONS USED IN THE STUDY

A.1 hasTwoDigits

A.1.1 Problem Statement. Write a function that determines whether the given positive number has exactly two digits.

A.1.2 Solution versions.

Student solution

```
def hasTwoDigits(x):
    if x >= 10:
            return True
```

Valid generated correction

```
def hasTwoDigits(x):
    if 10 <= x <= 99:
        return True
    else:
        return False
```

Problematic generated correction

```
def hasTwoDigits(x):
    if 1 <= x//10 < 10:
            return True
    else:
        return False
```

A.1.3 Bugs in student code.

(1) No upper bound check for whether number has more than two digits
(2) Returns None instead of False when two-digit test fails

A.1.4 Size of issue. Small

A.1.5 Problems with problematic solution.

(1) Esoteric math in if comparison: not wrong, but unnecessary and hard to understand

A.2 secondHalf

A.2.1 Problem Statement. Given a list l, return the second half of that list. If the list has an odd number of elements, include the middle element.

A.2.2 Solution versions.

Student solution

```
def secondHalf(l):
    lst=[]
    for i in range(len(l)):
        if i>=(len(l)-1)//2:
            lst.append(l[i])
    return lst
```

Valid generated correction

```
def secondHalf(l):
    lst=[]
    for i in range(len(l)):
        if i>=len(l)//2:
            lst.append(l[i])
    return lst
```

Problematic generated correction

```
def secondHalf(l):
    lst = len(l)
    for i in range(len(l)):
        if i >= 0:
            return l[lst // 2:lst]
    return l[(lst - 1) // 2:lst]
```

A.2.3 Bugs in student code.

(1) off-by-one error in if condition: `-1` is unnecessary. Adds an extra element in even cases.

A.2.4 Size of issue. Small

A.2.5 Problems with problematic solution.

(1) Extremely esoteric sequence of code execution: always returns the second time through the loop, second return statement never happens and wouldn't return the correct result, loop and if are unnecessary for the actual logic of the solution to work
(2) The variable name `lst` is not appropriate for how it's used in the code

A.3 listOfLists

A.3.1 Problem Statement. Given a list of lists, return a new (1D) list that contains all of the elements present in the original lists, with no duplicates. This single list should be sorted according to the built-in Python sort method. Hint: This problem becomes fairly simple if you use sets!

A.3.2 Solution versions.

Student solution

```
def listOfLists(l):
    listSet = set()
    lst = []
    for i in range(len(l)):
        for k in range(len(l[i])):
            listSet.add(l[i][k])
    for num in listSet:
        lst += [num]
```

```
    return lst
```

Valid generated correction

```
def listOfLists(l):
    listSet = set()
    lst = []
    for i in range(len(l)):
        for k in range(len(l[i])):
            listSet.add(l[i][k])
    for num in listSet:
        lst += [num]
    return sorted(lst)
```

Problematic generated correction

```
def listOfLists(l):
    listSet = set()
    for lst in l:
        for elem in lst:
            listSet.add(elem)
    lst = []
    for num in listSet:
        lst += [num]
    return sorted(list(listSet))
```

A.3.3 Bugs in student code.

(1) Did not sort the list before returning (forgot part of problem statement?)

A.3.4 Size of issue. Small

A.3.5 Problems with problematic solution.

(1) Unnecessarily moves code around
(2) Computes `lst`, then doesn't use it - uses `list(listSet)` directly instead

A.4 howManyEggCartons

A.4.1 Problem Statement. Given a number of eggs (as an integer), return the number of egg cartons needed to hold that many eggs. Cartons hold 12 eggs each, so from 1 to 12 eggs requires one carton, 13 to 24 requires two, etc.

A.4.2 Solution versions.

Student solution

```
def howManyEggCartons(eggs):
    quad = eggs // 12
    if quad % 2 != 0:
        return quad + 1
    else:
        return quad
```

Valid generated correction

```
def howManyEggCartons(eggs):
    quad = eggs // 12
    if eggs % 12 != 0:
        return quad + 1
    else:
        return quad
```

Problematic generated correction

```
def howManyEggCartons(eggs):
    quad = eggs // 12
    if eggs % 2 != 0:
        return quad + 1
    else:
        return quad
```

A.4.3 Bugs in student code. Comparing the wrong quantities in if modulo expression:

(1) quad instead of eggs
(2) 2 instead of 12

A.4.4 Size of issue. Small

A.4.5 Problems with problematic solution.

(1) Incorrect solution: `%2` instead of `%12` (passes unit tests by chance)

A.5 nearestBusStop

A.5.1 Problem Statement. Write a function that takes a non-negative street number and returns the nearest bus stop to the given street. Buses stop every 8th street, including street 0, and ties go to the lower street, so the nearest bus stop to 12th street is 8th street, and the nearest bus stop to the 13th street is 16th street.

A.5.2 Solution versions.

Student solution

```
def nearestBusStop(street):
    if street % 8 == 0:
        stop = street
    if street % 8 <= 4:
        stop = street
    if street % 8 > 4:
        stop = street +1
    return stop
```

Valid generated correction

```
def nearestBusStop(street):
    if street % 8 == 0:
        stop = street
    if street % 8 <= 4:
        stop = street - street % 8
    if street % 8 > 4:
        stop = street - street % 8 + 8
    return stop
```

Problematic generated correction

```
def nearestBusStop(street):
    if street % 8 > 4:
        stop = street
    if street % 8 > 4:
        return street + 8 - street % 8
    else:
        return street - street % 8
```

A.5.3 Bugs in student code.

(1) Incorrect computation for nearest bus stop in "rounding down" case

(2) Incorrect computation for nearest bus stop in "rounding up" case

A.5.4 Size of issue. Large

A.5.5 Problems with problematic solution.

(1) First if is extraneous and doesn't affect output
(2) Too many unnecessary changes to the student code

A.6 anyLowercase

A.6.1 Problem Statement. Given a string s, return True if any character in that string is lowercase (between 'a' and 'z'), and False otherwise.

A.6.2 Solution versions.

Student solution

```
import string
def anyLowercase(s):
    for i in range(len(s)):
        if (s[i] not in
            string.ascii_lowercase):
            return False
    return True
```

Valid generated correction

```
import string
def anyLowercase(s):
    for i in range(len(s)):
        if (s[i] in
            string.ascii_lowercase):
            return True
    return False
```

Problematic generated correction

```
import string
def anyLowercase(s):
    for i in s:
        if (i in
            string.ascii_lowercase):
            return True
    return False
```

A.6.3 Bugs in student code.

(1) Conceptual issue: returns True if and only if *all* lowercase, instead of *any*. (Problem with interpreting problem statement?)

A.6.4 Size of issue. Large

A.6.5 Problems with problematic solution.

(1) Unnecessary change: iterating through items instead of indices

A.7 middleElement

A.7.1 Problem Statement. Given a non-empty list, l, return the middle element of that list. If the list has an even number of elements, return the middle-right element.

A.7.2 Solution versions.

Student solution

```
def middleElement(l):
    if len(l)-1%2==0:
        return len(l)-1%2
    else:
        return len(l)-1//2 +1
```

Valid generated correction

```
def middleElement(l):
    if (len(l)-1)%2==0:
            mid_index = (len(l)-1)//2
    else:
            mid_index = (len(l)-1)//2 +1
    return l[mid_index]
```

Problematic generated correction

```
def middleElement(l):
    length = len(l)
    if len(l) / (1 % 2) == 0:
        return len(l) - 1 % 2
    else:
        return l[length // 2]
```

A.7.3 Bugs in student code.

(1) Missing parentheses (3 separate times)
(2) `%2` instead of `//2` in if body (copy-paste issue?)
(3) Returns index, not element

A.7.4 Size of issue. Large

A.7.5 Problems with problematic solution.

(1) Esoteric and unnecessary if statement: it's almost always false, and would return the wrong result if true (e.g. with list of length 0)
(2) `length` variable is used inconsistently, and is arguably unnecessary

A.8 sumOfDigits

A.8.1 Problem Statement. Given a number n, return the sum of n's digits.

A.8.2 Solution versions.

Student solution

```
def sumOfDigits(n):
    i=0
    summation=0
    while i<=nsize(n):
        r=n//(10**(n-i-1))
        remain= r%10
        summation+=remain
    return summation
```

Valid generated correction

```
def sumOfDigits(n):
    i=0
    summation=0
    while i<len(str(n)):
```

```
        r=n//(10**(len(str(n))-i-1))
        remain= r%10
        summation+=remain
        i+=1
    return summation
```

Problematic generated correction

```
def sumOfDigits(n):
    i = abs(n)
    summation = 0
    while n > 0:
        r = n % 10
        n = n // 10
        summation += r
    return summation
```

A.8.3 Bugs in student code.

(1) `nsize(n)` should be an expression which evaluates to number of digits in `n`, e.g. `len(str(n))`
(2) When calculating `r`, `n` should also be replaced with the number of digits in `n`
(3) Loop doesn't terminate because `i` doesn't update
(4) If loop did terminate (by starting `i` at 0 and incrementing by 1), it would iterate one more time than needed

A.8.4 Size of issue. Large

A.8.5 Problems with problematic solution.

(1) Line 2 (`i = abs(n)`) is extraneous, the value is never used
(2) Doesn't preserve student intent on how to iterate through digits and extract them

B PRE-SURVEY

The pre-survey given to participants consisted of two multiple-choice questions.

B.1 Question 1. Have you ever been a Teaching Assistant for a CSE course before? If you have, then for how many total semesters?

- 0: I haven't been a TA yet
- 1: I have been a TA once
- 2: I have been a TA for two semesters
- >2: I have been a TA for more than two semesters

B.2 Question 2. How much experience do you have with Python?

- None: never tried it
- A little: I have used it once or twice
- Some: I have used it before, but am not extremely comfortable with it
- Lots: I am quite comfortable with Python

C POST-SURVEY

The post-survey given to participants consisted of two free text questions.

C.1 Question 1. What was the most helpful part of the interface as you were trying to identify the problems with the student code?

C.2 Question 2. What was the hardest or most cumbersome part of the task?